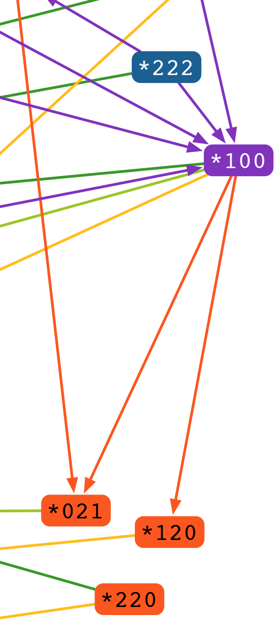
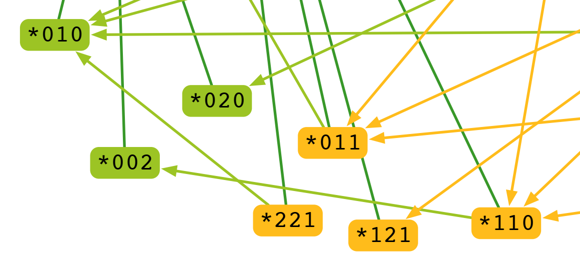




Synthesizing fault-tolerant distributed algorithms

Danny Dolev
Hebrew University of Jerusalem

Christoph Lenzen
MPI Saarbrücken

Keijo Heljanko
Joel Rybicki
Jukka Suomela
Siert Wieringa
Aalto University & HIIT

Janne H. Korhonen
Matti Järvisalo
University of Helsinki & HIIT

Ulrich Schmid
TU Wien

Computational Logic Day · Aalto University · 16 December 2014

What is this talk about?

Developing fault-tolerant distributed algorithms for consensus-like problems using *computational techniques*.

Verification vs synthesis

Verification:

*“**Check** that given A satisfies the specification S.”*

Synthesis:

*“**Construct** an A that satisfies a specification S.”*

The model problem

The *synchronous counting problem*:

- Closely related to consensus
- Self-stabilization
- Byzantine fault tolerance
- Hard to come up with correct algorithms

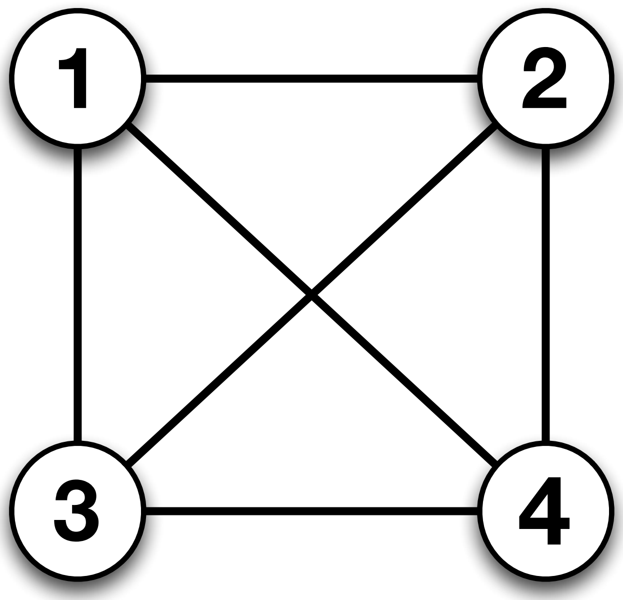
Our work

Prior work: Are there **efficient** and **compact** deterministic algorithms?
Dolev et al. (**SSS 2013**)

Recent work: Developing and evaluating different **synthesis techniques**

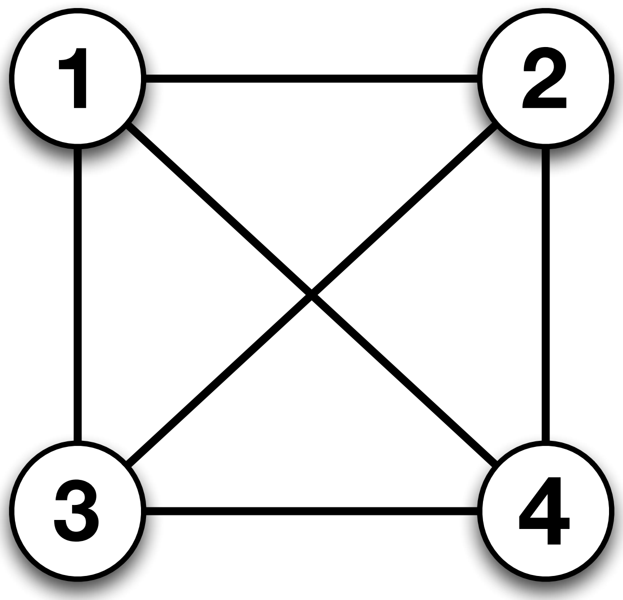
Synchronous counting

The model



- *n* processors
- *s* states per node
- arbitrary initial state

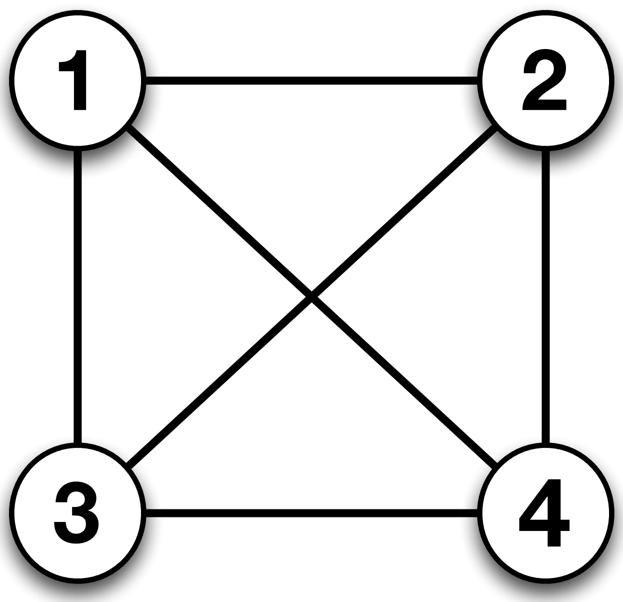
The model



- *n* processors
- *s* states per node
- arbitrary initial state

Synchronous step:
1. send state to all neighbors
2. update state

The model



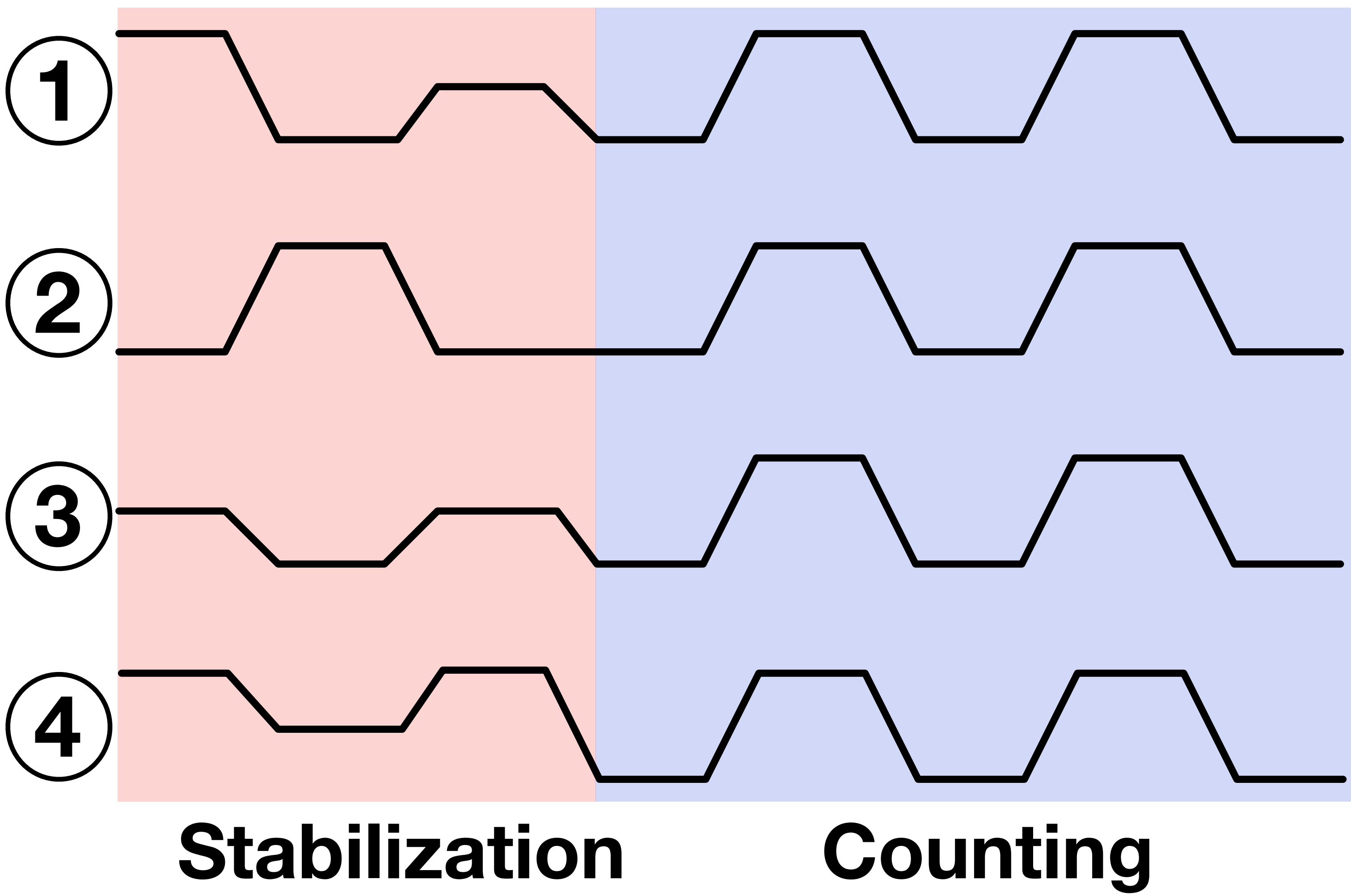
- *n* processors
- *s* states per node
- arbitrary initial state

Synchronous step:

1. send state to all neighbors
2. update state

algorithm
=
transition function

Self-stabilizing counting

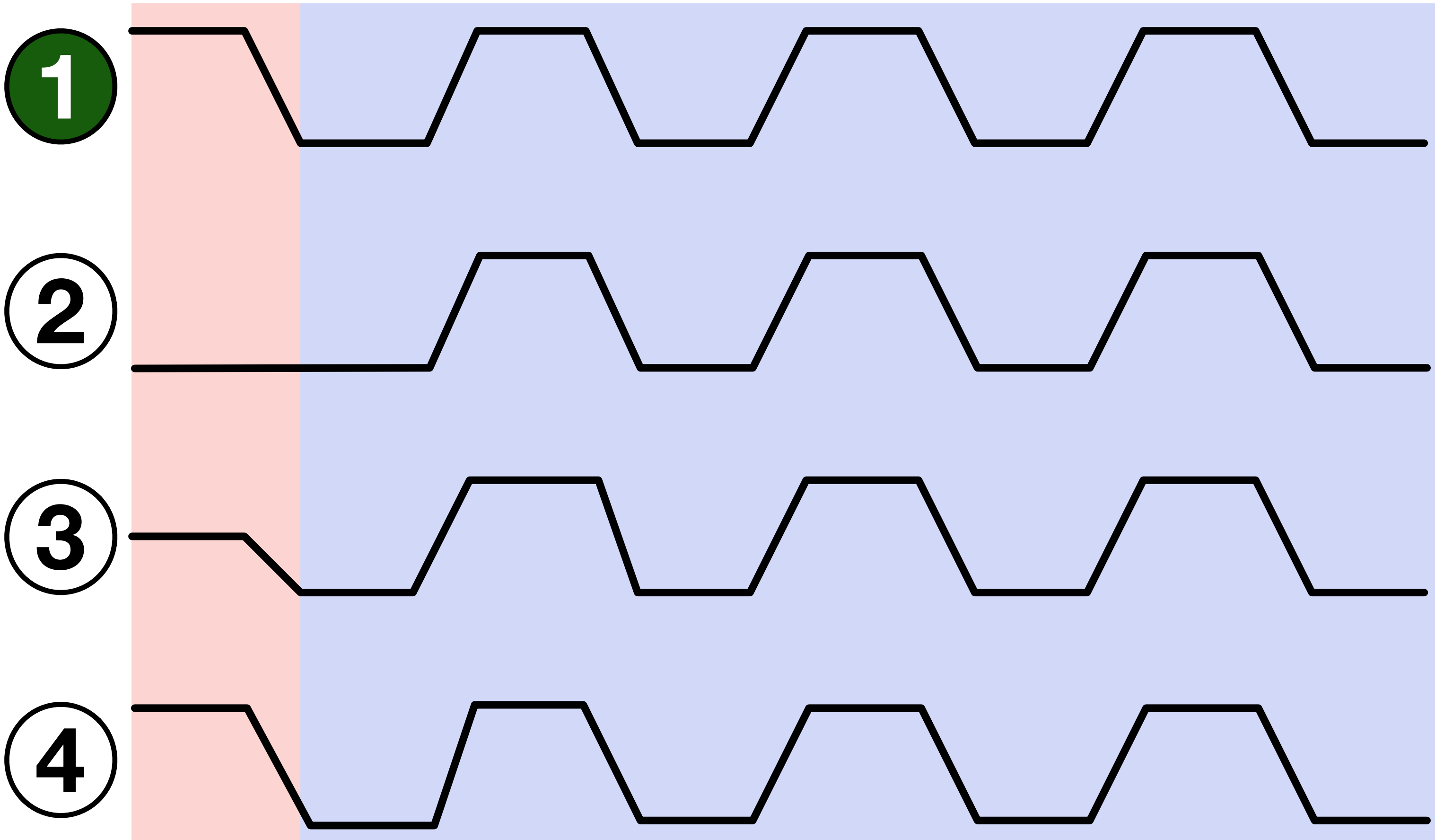


Self-stabilizing counting

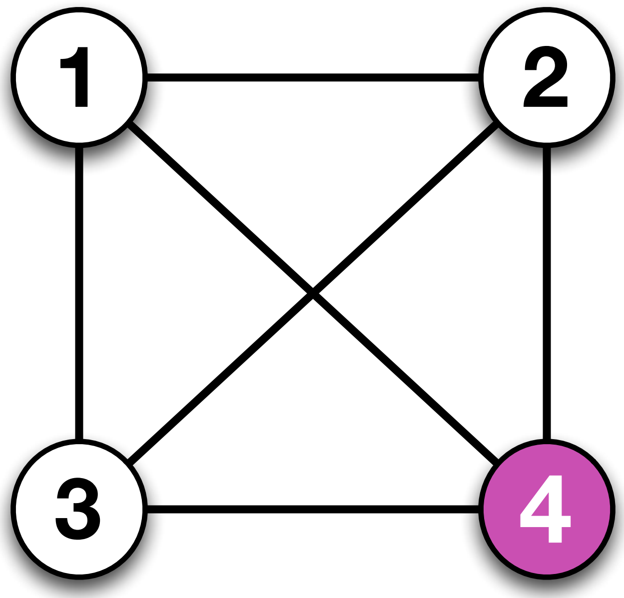
A simple algorithm solves the problem

Self-stabilizing counting

Solution: Follow the leader.

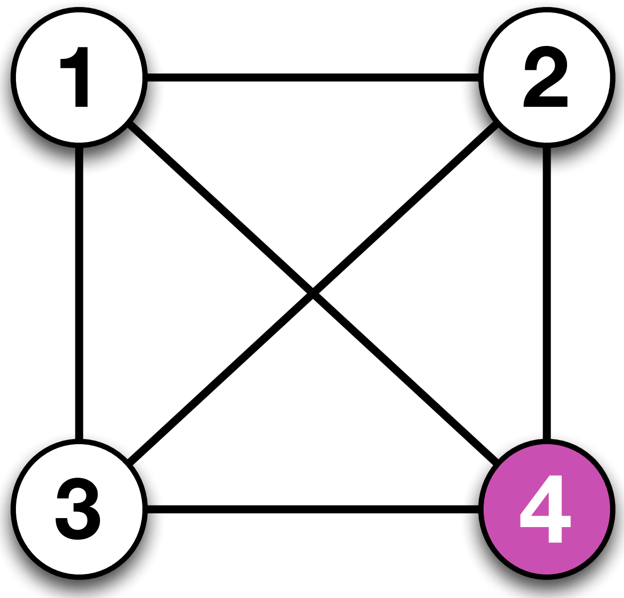


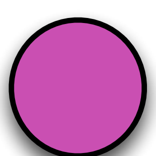
Tolerating Byzantine failures



Assume that at most *f* nodes may be **Byzantine**.

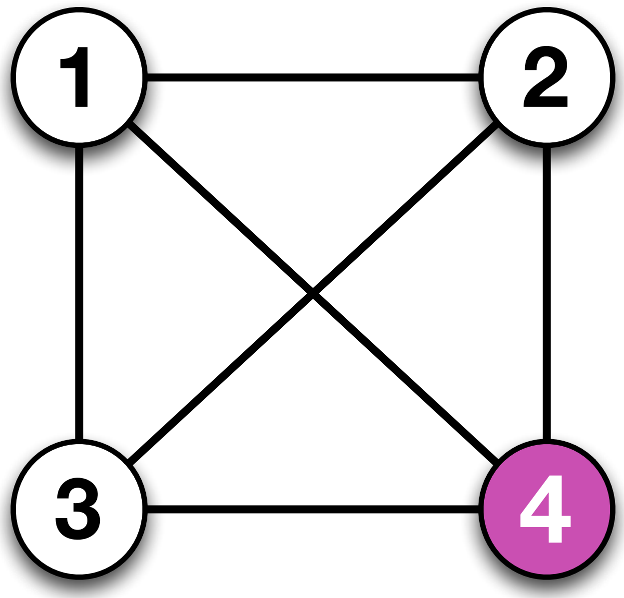
Tolerating Byzantine failures

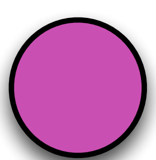




can send *different* messages to non-faulty nodes!

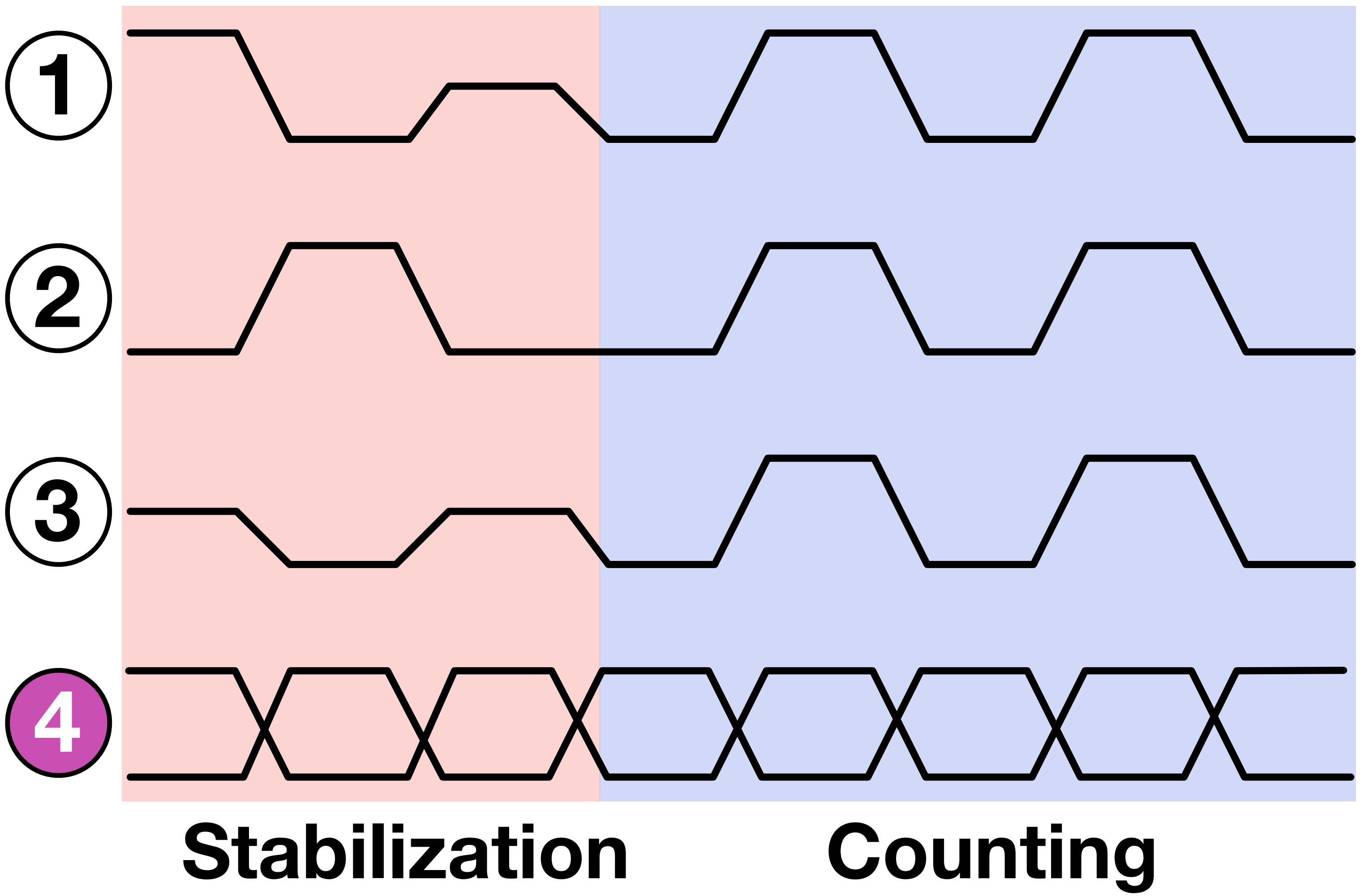
Tolerating Byzantine failures



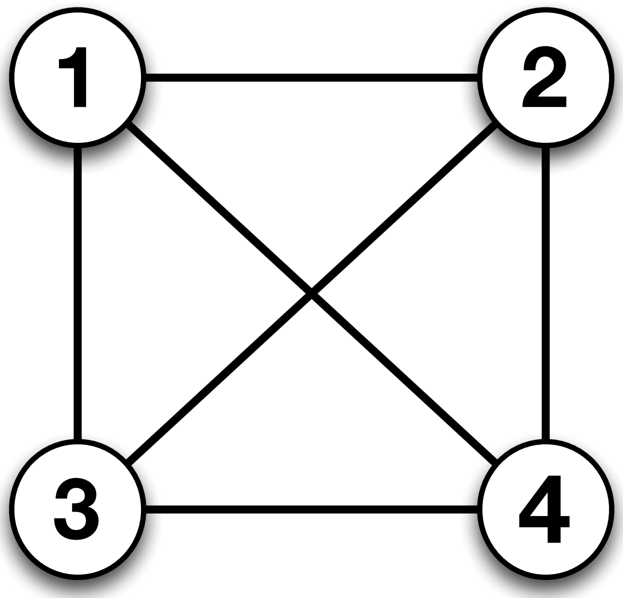

can send *different* messages to non-faulty nodes!

Note: Easy if self-stabilization is not required!

Fault-tolerant counting



The model with failures



- *n* processors
- *s* states
- arbitrary initial state
- at most *f* Byzantine nodes

Some basic facts

- How many states (per node) do we need?
  - $s \geq 2$
- How many faults can we tolerate?
  - $f < n/3$
- How fast can we stabilize?
  - $t > f$

Pease et al., 1980
Fischer & Lynch, 1982

Solving synchronous counting

Deterministic solutions with large s known for similar problems (e.g. D. Dolev & Hoch, 2007)

Randomized solutions for counting with small s and large t in expectation (e.g. S. Dolev: Self-stabilization)

We have synthesized *deterministic* algorithms with small s and t for the case $f = 1$ (**SSS '13**)

Finding an algorithm

The size of the search space is s^b where $b = ns^n$.

parameters	search space
$n = 4$ $s = 2$	$2^{64} \approx 10^{19}$

Finding an algorithm

The size of the search space is s^b where $b = ns^n$.

parameters	search space
n = 4 s = 2	$2^{64} \approx 10^{19}$
n = 4 s = 3	$3^{324} \approx 10^{154}$

Main results, $f = 1$

If $4 \leq n \leq 5$:

- **lower bound:** no 2-state algorithm
- **upper bound:** 3 states suffice

If $n \geq 6$:

- 2 states always suffice

Synthesis techniques

Cut-off result

For any fixed *s*, *f* and *t*:

There is an algorithm **A** for *n* nodes

$$\Downarrow$$

There is an algorithm **B** for *n*+1 nodes with same *s*, *f* and *t*

Direct encoding

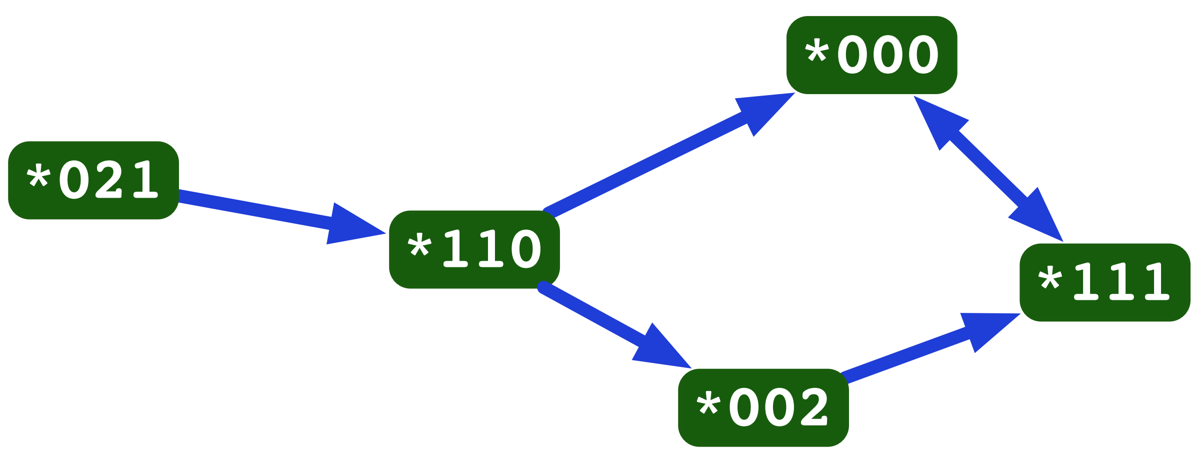
- Fix *n*, *s* and *f*
- The existence of an algorithm is a finite combinatorial decision problem
- Apply **SAT solvers**

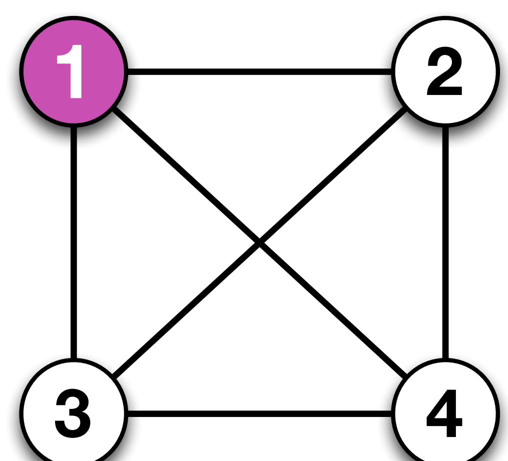
Verification is easy

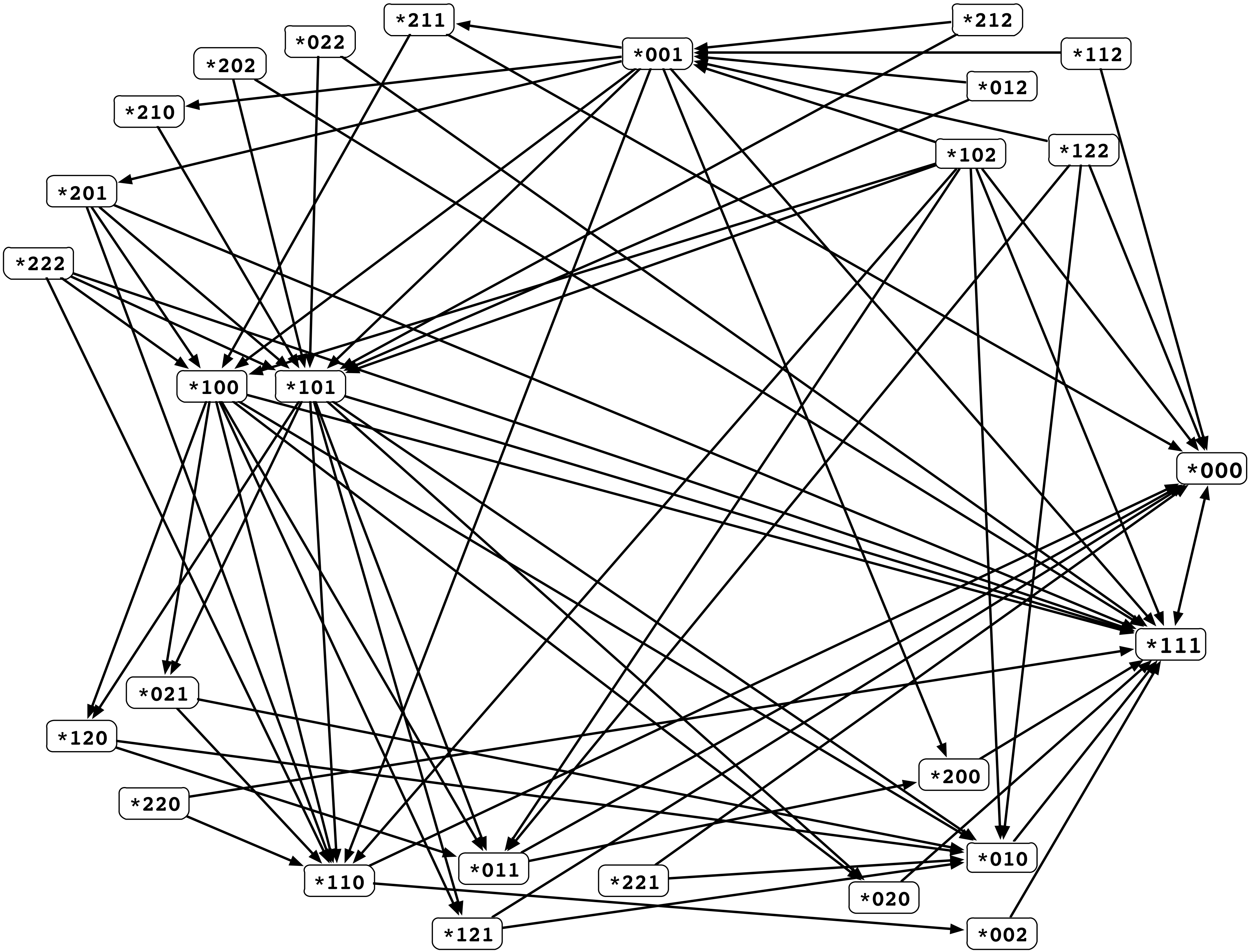
- Let *F* be a set of faulty nodes, $|F| \le f$
- Construct a *state graph* G_F from **A**:

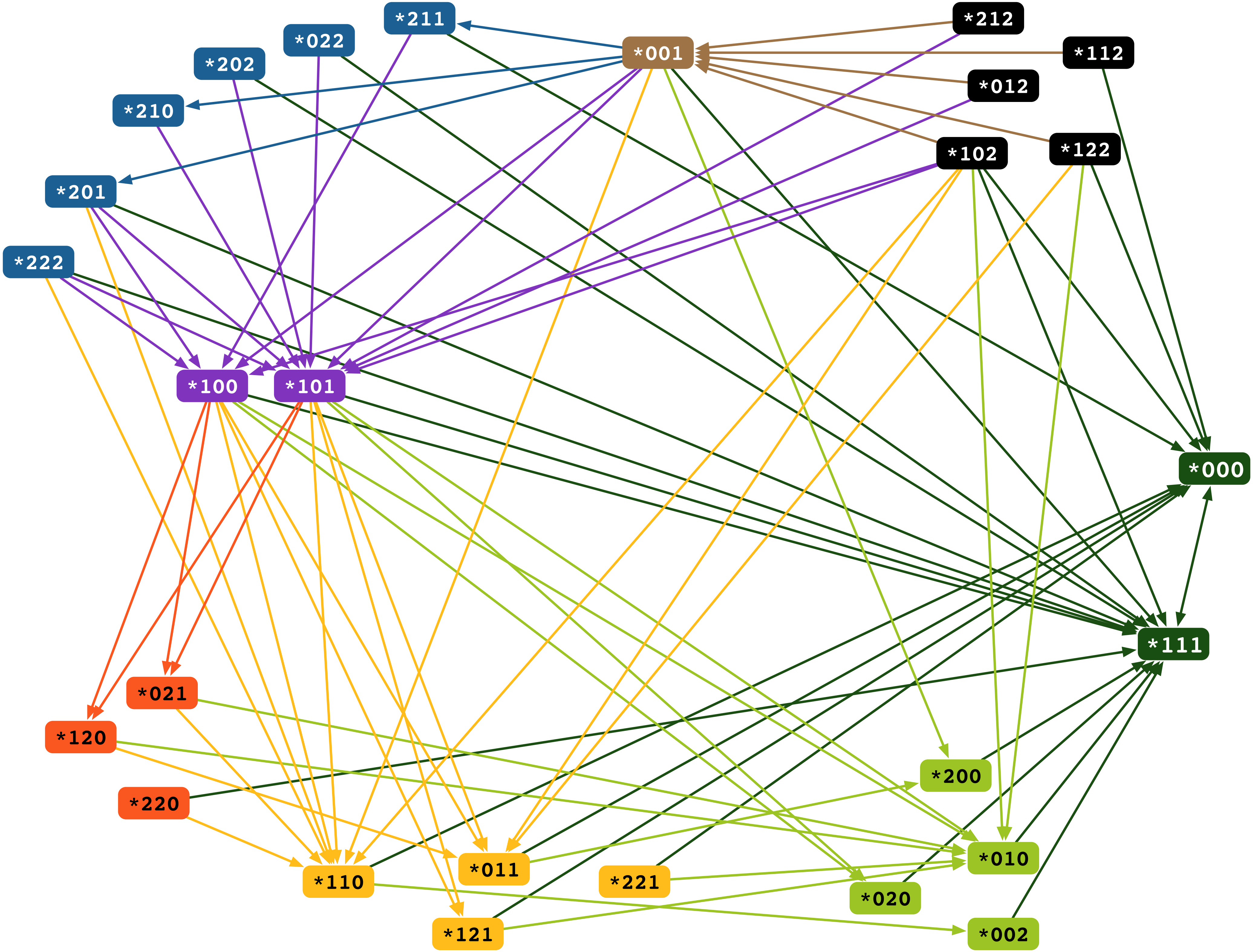
  Nodes = actual states

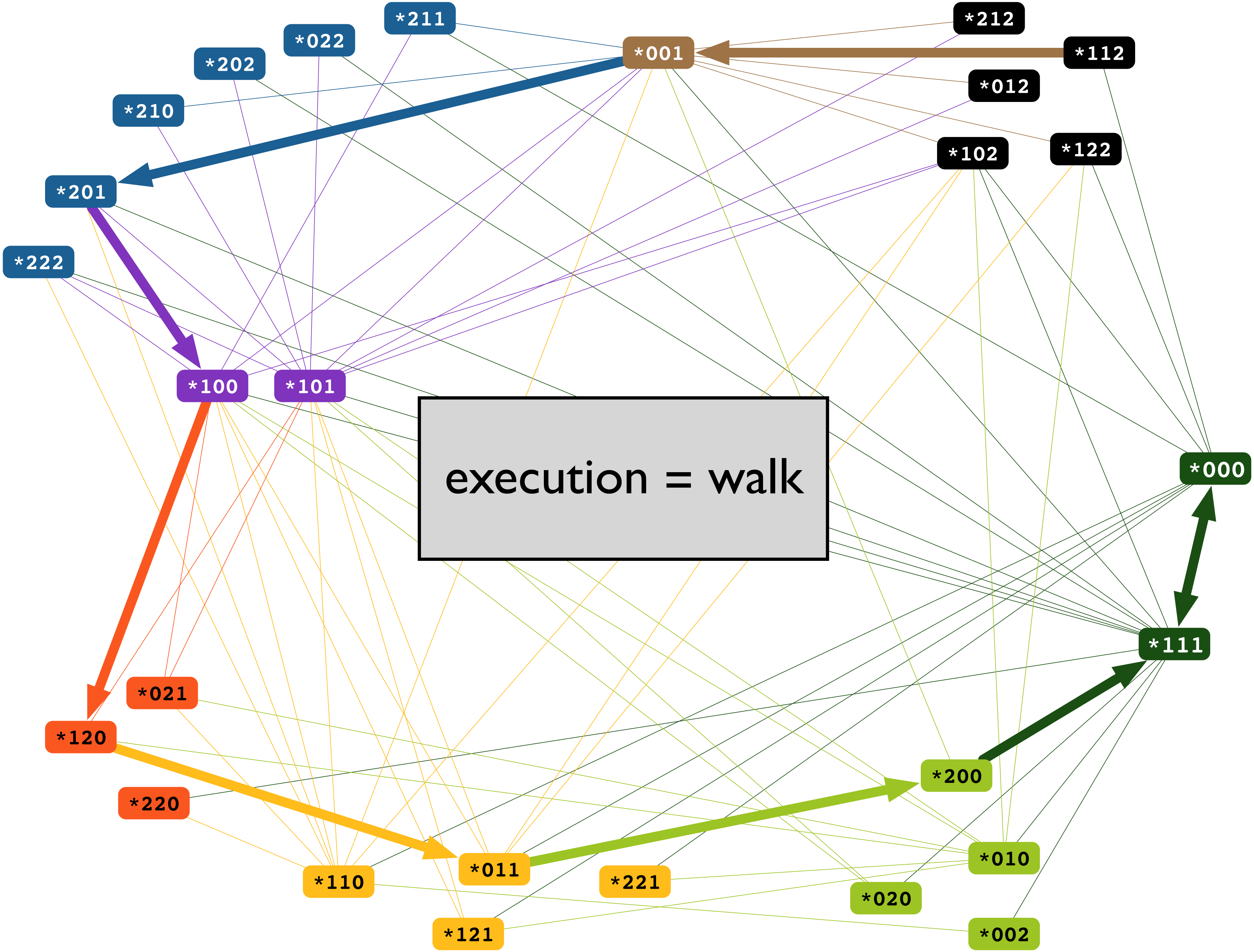
  Edges = possible state transitions



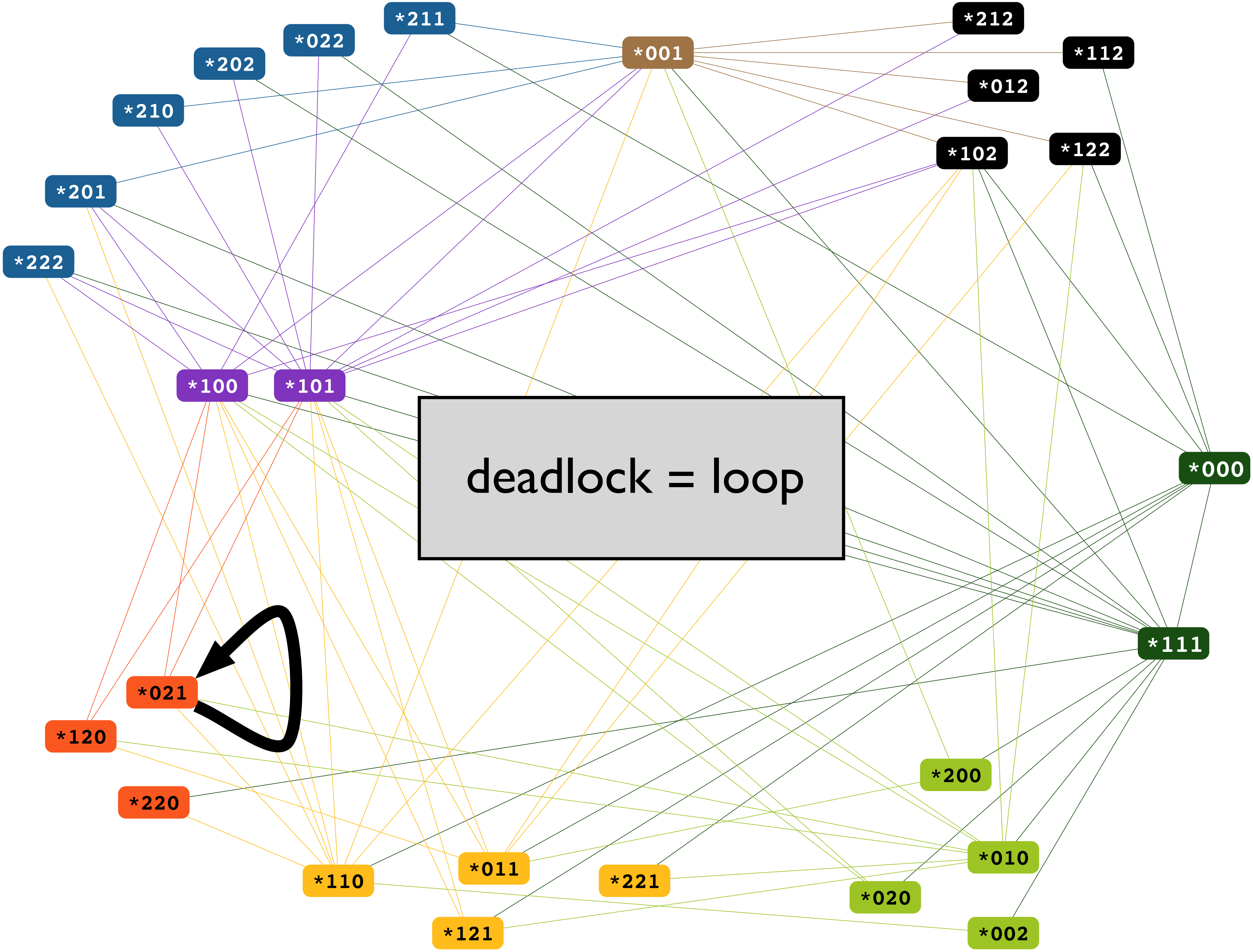


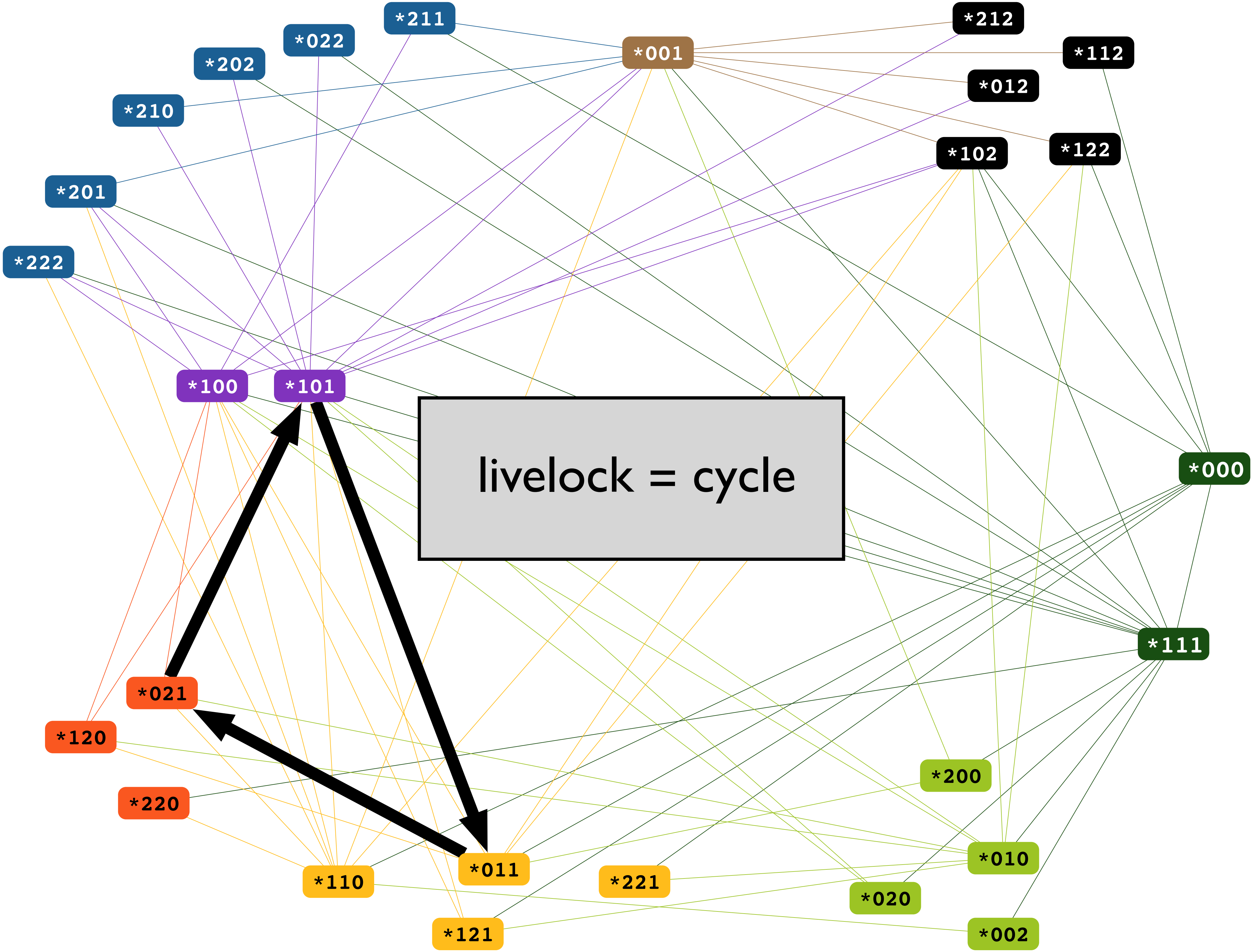

*211
*212
*022
*001
*112
*202
*012
*210
*102
*122
*201
*222
*100
*101
*000
*111
*021
*120
*200
*220
*010
*011
*221
*110
*020
*121
*002

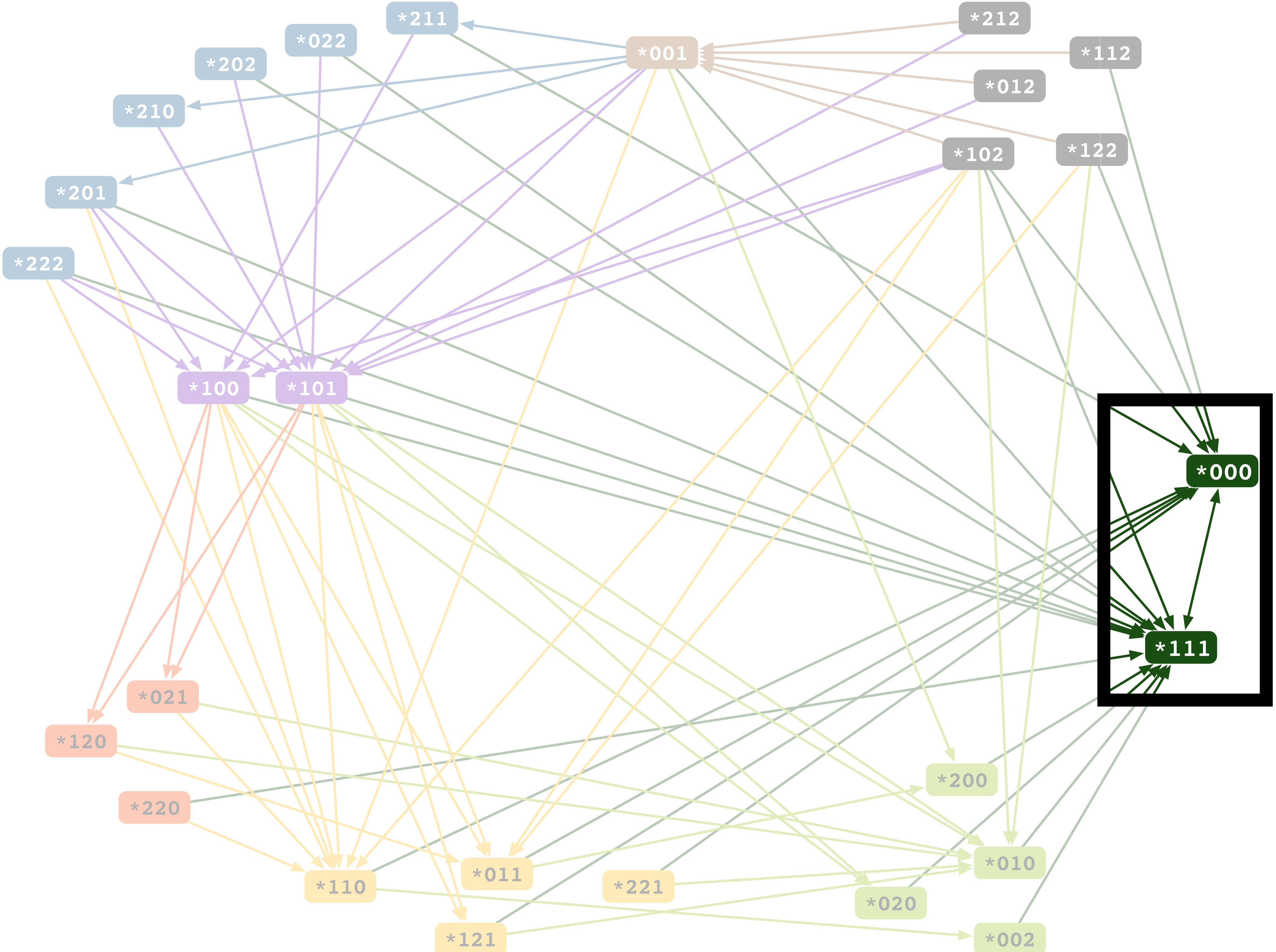

*211
*022
*202
*210
*201
*222
*001
*212
*112
*012
*102
*122
*100
*101
*000
*111
*021
*120
*220
*200
*010
*020
*002
*110
*011
*221
*121

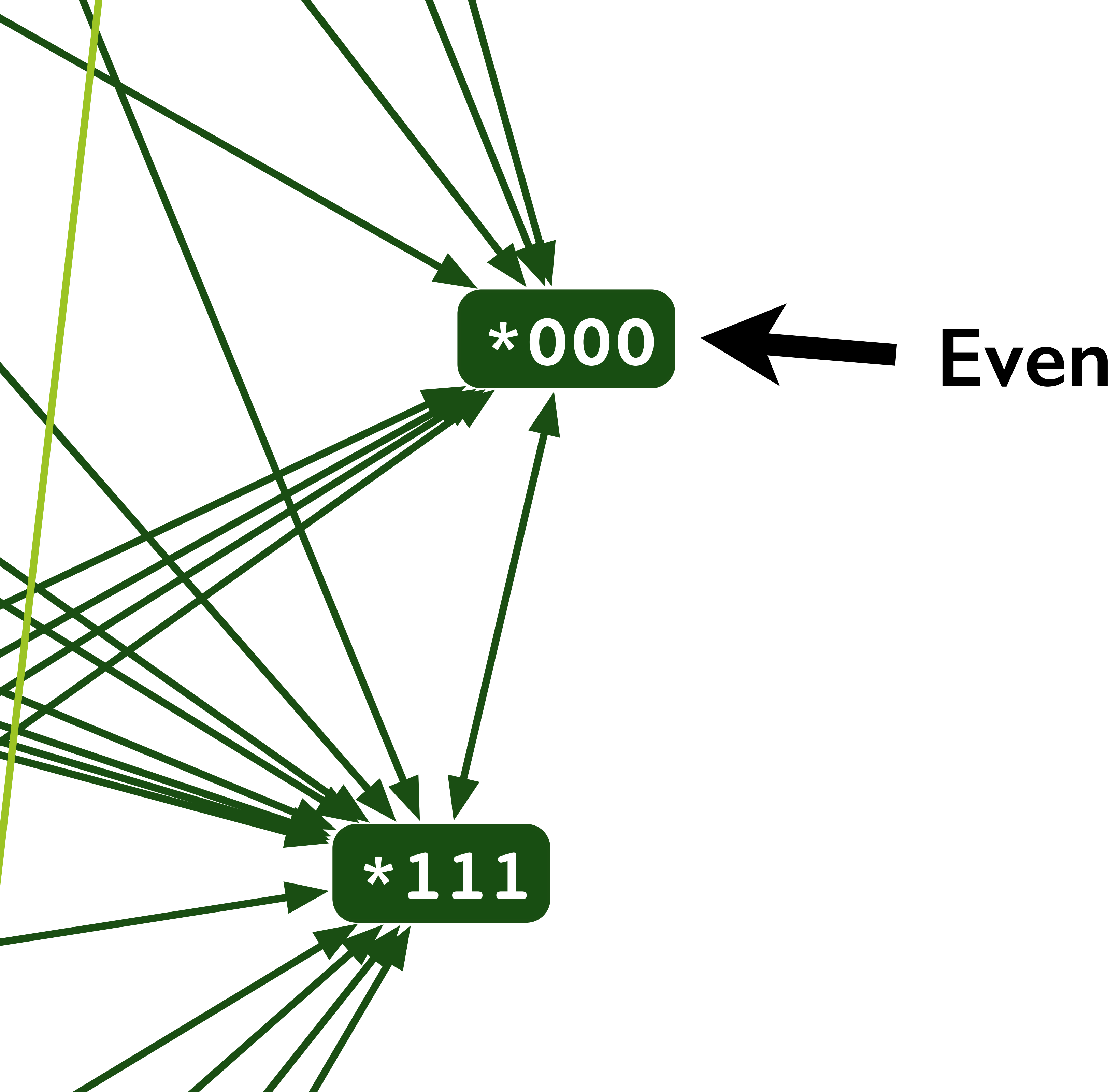


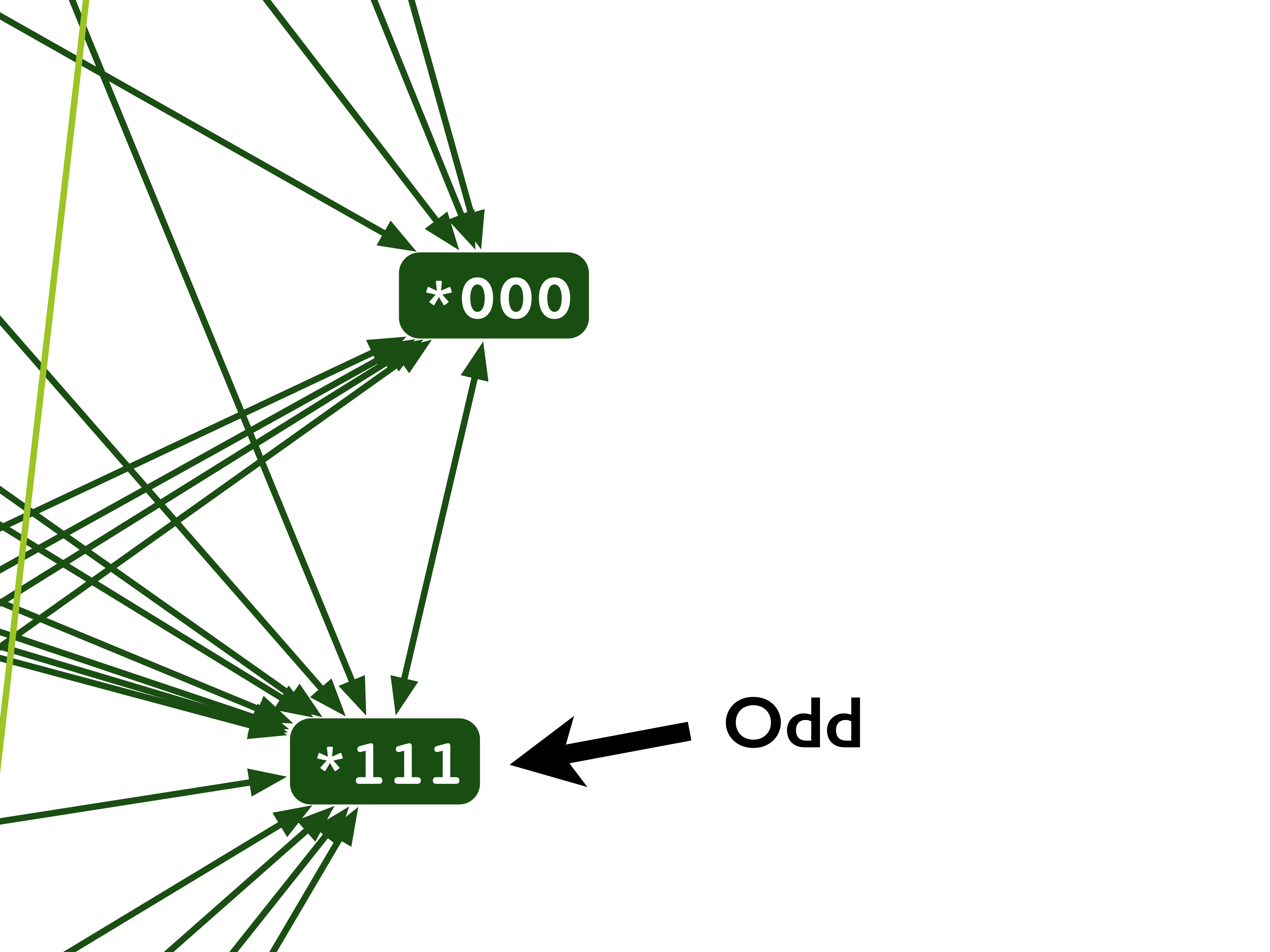
*211
*022
*202
*210
*001
*212
*112
*012
*102
*122
*201
*222
*100
*101
execution = walk
*000
*111
*021
*120
*220
*200
*010
*110
*011
*221
*020
*121
*002


*211
*022
*202
*210
*201
*222
*001
*212
*112
*012
*102
*122
*100
*101
deadlock = loop
*000
*111
*021
*120
*220
*200
*010
*110
*011
*221
*020
*121
*002


*211
*022
*202
*210
*201
*222
*001
*212
*112
*012
*102
*122
*100
*101
livelock = cycle
*000
*111
*021
*120
*220
*110
*011
*121
*221
*200
*010
*020
*002


*211
*022
*202
*210
*201
*222
*001
*212
*112
*012
*102
*122
*100
*101
*000
*111
*021
*120
*220
*200
*010
*110
*011
*221
*020
*121
*002


*000
Even
*111


*000
*111
Odd

Verification is easy

A is **correct**	$\Leftrightarrow$	Every G_F is **good**
no deadlocks	$\Leftrightarrow$	G_F is loopless
stabilization	$\Leftrightarrow$	All nodes have a path to **0**
counting	$\Leftrightarrow$	$\{\mathbf{0},\mathbf{1}\}$ is the only cycle

From verification to synthesis

The encoding uses the following variables:

$$x_{i,u,s} \quad \Leftrightarrow \quad A_i(u) = s$$

$$e_{q,r} \quad \Leftrightarrow \quad \text{edge } \ (q, r) \ \text{ exists}$$

$$p_{q,r} \quad \Leftrightarrow \quad \text{path } \ q \rightsquigarrow r \ \text{ exists}$$

$$x_{i,u,s} \quad \Rightarrow \quad e_{q,r} \quad \Rightarrow \quad p_{q,r}$$

Direct encoding

- Solver is a black box: no domain-knowledge
- Relatively easy to setup
- Size of instances **blows up:**

instance: n, s, t	**variables**	**clauses**
4 3 10	6k	31k
5 3 10	45k	36k
6 3 10	403k	4M

Counter-example guided search

- A problem-specific synthesis algorithm
- CEGAR-inspired search
- Uses SAT solver to find ***counter-examples***
- Learn constraints on-the-fly

A high-level overview

While algorithm candidates exist:

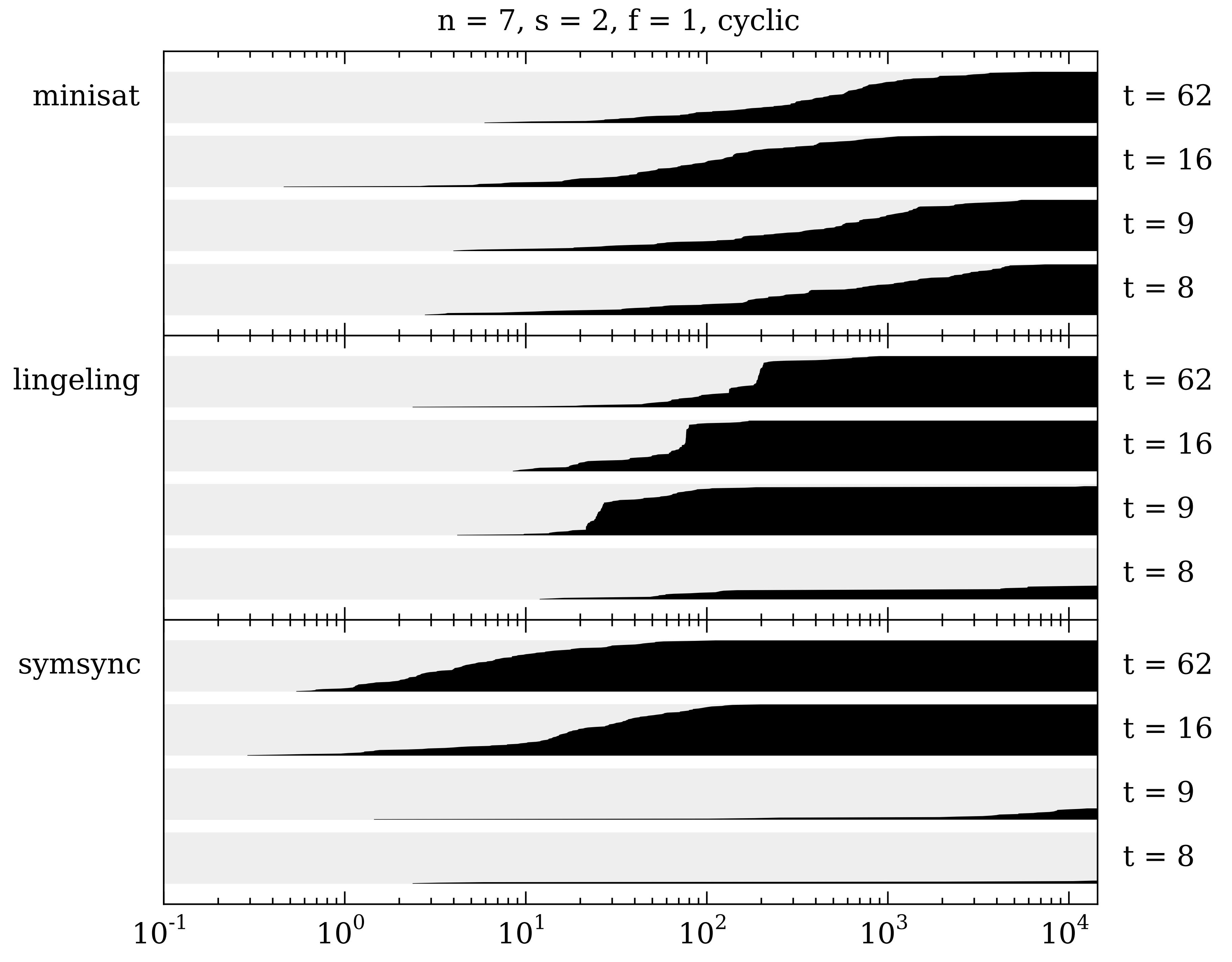
- Guess an algorithm **A**
- Use a SAT solver to check if **A** is correct
- If not, solver gives a counter-example. Learn new constraints that forbid bad algorithms

How to learn ***useful*** constraints from counter-examples?

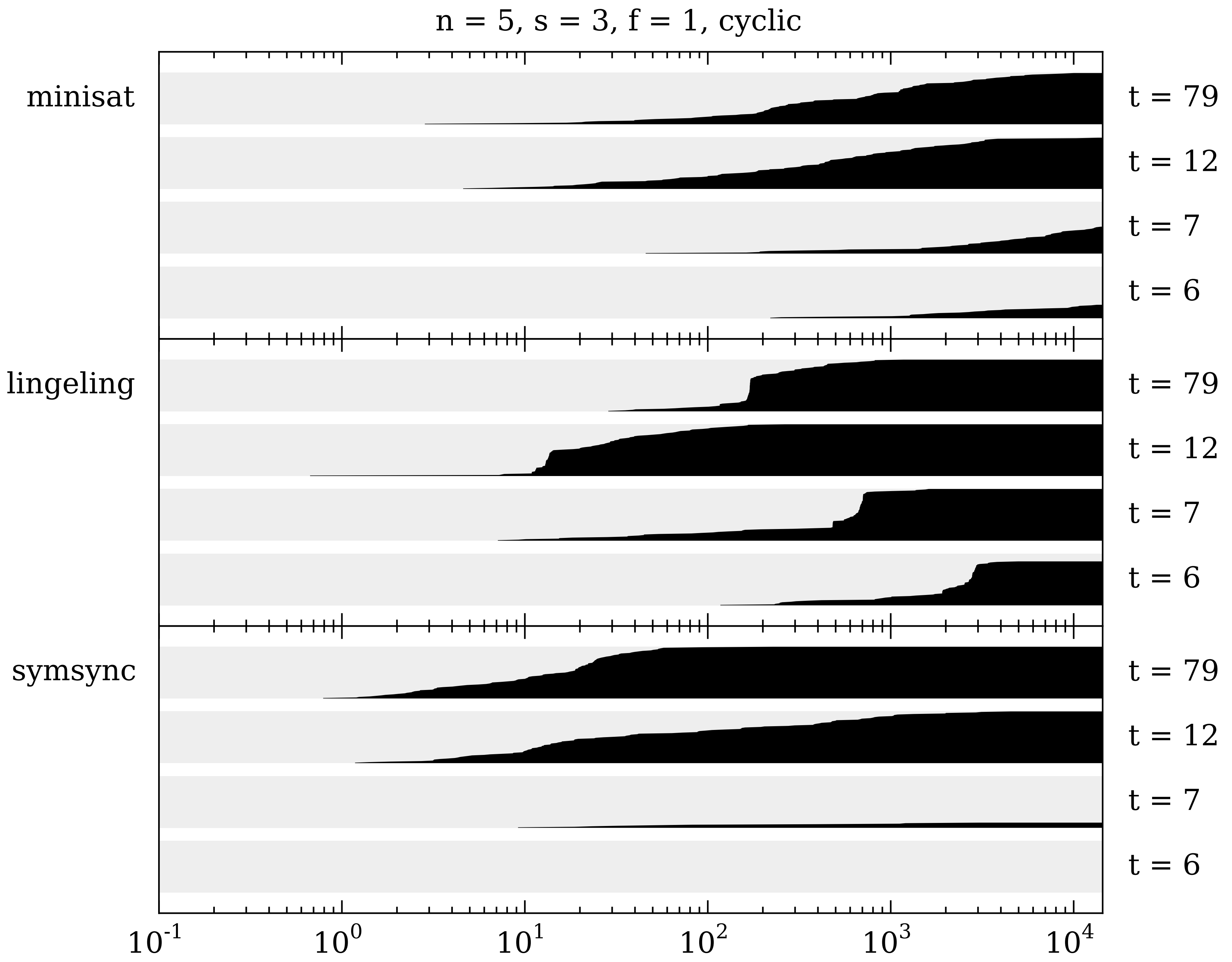
Some experiments

Experiment setup

- Direct encoding:
  MiniSAT and lingeling solvers
- 'symsync': the guided search algorithm
- same instance on 100 processors in parallel,
  different random seeds



n = 7, s = 2, f = 1, cyclic
minisat
lingeling
symsync
t = 62
t = 16
t = 9
t = 8
t = 62
t = 16
t = 9
t = 8
t = 62
t = 16
t = 9
t = 8
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}


n = 5, s = 3, f = 1, cyclic
minisat
lingeling
symsync
t = 79
t = 12
t = 7
t = 6
10^-1
10^0
10^1
10^2
10^3
10^4

Summary

- Synthesis a tool for *theory of distributed computing*
- Results: optimal fault-tolerant algorithms
- Complementary approaches for fast synthesis

Summary

- Synthesis a tool for *theory of distributed computing*
- Results: optimal fault-tolerant algorithms
- Complementary approaches for fast synthesis

Synthesis in our other work:

- local graph coloring
- finding large cuts **arXiv:1402.2543**

Thanks!